Plastic: Project Proposal

Name

Plastic

Repo Name

plastic

Description

This is a "translation by intent" facility that provides leverage in the model-to-model transform space. The model-to-model translation problem is pervasive in writing SDN controller applications, both internally and in supporting micro services. By convention, Plastic emphasizes writing translations that would be as resilient to model changes as possible. This is implemented through various mechanisms from declarative expressions of schemas, through schema semi-independent morphing, up to unbounded data morphing.
[[File:UpstreamingPlastic.zip|thumbnail]]

What is a mapping problem?

* Occurs in internals of a system behind endpoints
* ODL context – moving from northbound to southbound representations
* Sometimes need to trivially convert data representation
* JSON, XML, YML, other parse-able formats
* Sometimes need to change abstractions (1:1, N:1, N:N)
* Morph one model into completely different model
* Morph N models into one model

ODL/plastic Advantages

* Pay-as-you-go for complexity (field deployable changes)
* Declarative representations are emphasized (clear schemas)
* Translation-by-intent (say what you want, not how to do it)
* Deeper levels of abstraction to help keep custom logic schema-independence
* Can specify arbitrary morphing via plug-ins in JVM language
* Understands breaking large mapping problems up (both time and space) into small chunks

Solves problems like…

* Schema changes for device configurations across releases
* No more hard-wired dependency on vendor libraries
* In-the-field updating to support multiple versions of devices
* Light weight specifications avoid religiosity around “DRY"

Simplest Possible Plastic Mapping Specification

Input schema

* Exemplar-based specification (XML/JSON/…)
* Variables defining important values

Output schema

* Exemplar-based specification (XML/JSON/…)
* Variables defining bound values that are substituted

Input payload

* Same format of input schema
* Partial match required against input schema

Invoke

* "'Plastic.translate'"(payload, “ELN”, “1.0”, “JSON”, “AA”, “1.1”, “XML”)
* Schemas is an arbitrary file system hierarchy

Examples

* Example-1: no coding
* Example-2: least schema-dependent coding using "'morpher"' “plug-in”
* Example-3: highly dependent coding using "'classifier"' “plug-in”
* Translation pipeline showing how all “plug-in”s relate to the flow

Example-1: no coding

Input payload:

```
{
   "deviceName": "CXP-2501",
   "interfaceName": "TCP/1/0/24",
   "addresses": [
    {
       "address": "10.10.100.100",
       "prefix-length": "24"
     },
     {
       "address": "10.10.100.221",
       "prefix-length": "24"
     },
     {
       "address": "10.10.100.168",
       "prefix-length": "24"
     }
   ]
 }
```

Desired output payload:

```
 {
     "dev-name": "CXP-2501",
     "interface-name": "TCP/1/0/24",
     "objects": [
         {
             "subnet": "10.10.100.100/24"
         },
         {
             "subnet": "10.10.100.221/24"
         },
         {
             "subnet": "10.10.100.168/24"
         }
     ]
 }
```

Input schema:

```
{
   "deviceName": "${dName}",
   "interfaceName": "${iName}",
   "addresses": [
     {
       "address": "${addr[*]}",
       "prefix-length": "${pref-len[*]}"
     }
     ]
 }
```

Output schema:

```
{
   "dev-name": "${dName}",
   "interface-name": "${iName}",
   "objects": [
     {
       "subnet": "${addr[*]}/${pref-len[*]}"
     }
   ]
 }
```

Example-2: least schema-dependent coding using "'morpher"' “plug-in”

* Via simplest possible “morpher” plug-in
* Shows manipulation AFTER variable binding
* Morpher below is written in Java-style, not idiomatic Groovy...:

```
 class MyMorpher extends BasicMorpher
 {
         def tweakValues(Map inputs, Map outputs)
         {
                 outputs[‘ENDPOINT'] = inputs[‘ADDR-IN'] + ”:" + inputs[‘PORT-IN']
         }
 }
```

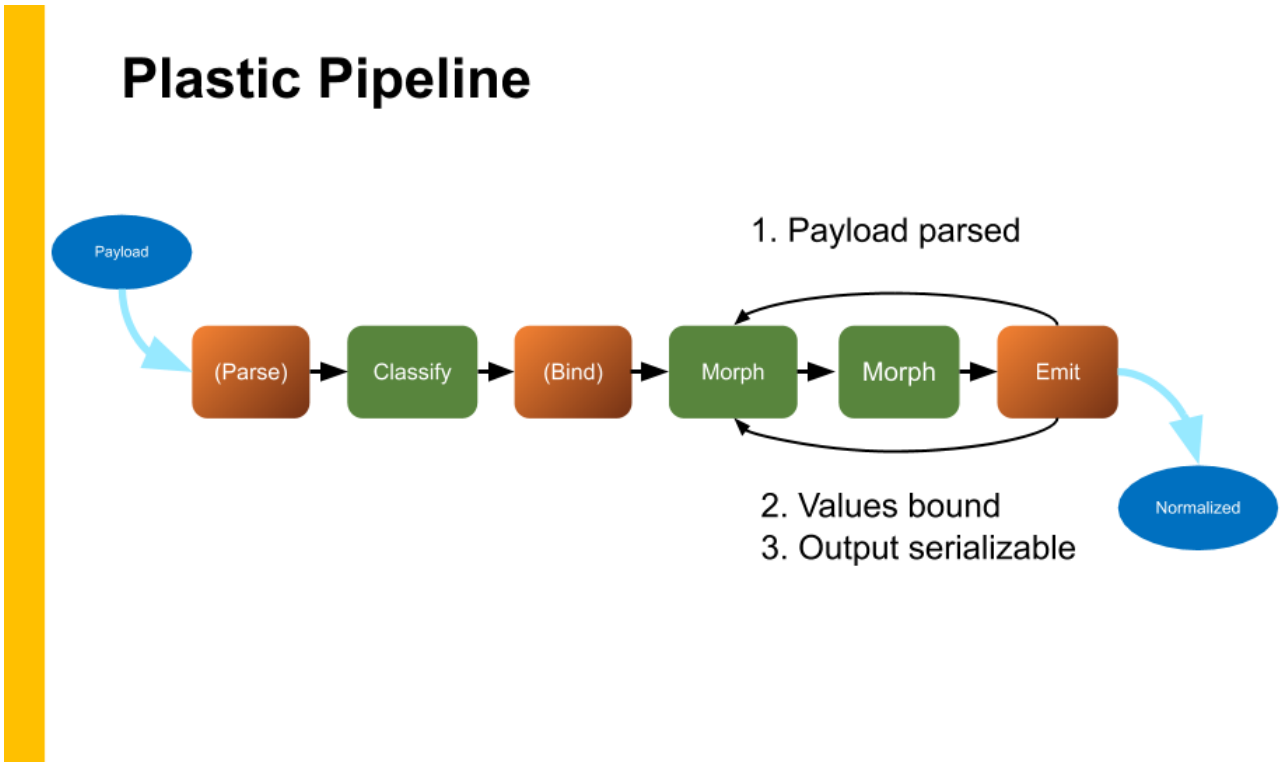
Example-3: highly dependent coding using "'classifier"' “plug-in”

* Via simplest possible “classifier” plug-in
* Shows just-in-time schema name resolution
* Payload is parsed JSON (ie, arrays/maps)
* Classifier below is written in Java-style, not idiomatic Groovy...:

```
class MyClassifier extends SimpleClassifier
 {
         String classify(Object parsedPayload) // payload is parsed JSON
         {
                 if (parsedPayload.astuff)
                         return “schema-A”
                 else
                         return “schema-B”
         }
 }
```

Translation Pipeline

* Optional user-supplied plug-ins are shown green



Extending Plastic

* Currently supports JSON and XML
* Easily extend to other text based formats like YML
* All formatting parsing, emitting, etc are defined behind an interface
* Possibility of processing binary data like NormalizedNode

More Good Stuff

* Default values (embed in schema, pass in on call, or ad hoc logic)
* Multiple morphers (clear separation of input and output schema logic)
* Optional run-time reload of schemas and translation logic
* Multithreaded “batch” processing for complex payloads
* Library implementation with simple synchronous API
* Command line runner for quick development
* Tutorials

Scope

Plastic is an independent library implemented in two JVM languages (Java and Groovy). In its initial inception, it is a library only and must be driven from client code. It is not Karaf-aware beyond having a jar/bundle as an artifact. It does not use any ODL constructs or services, so has no direct impact on any ODL components beyond its generic resource (cpu and memory) usage. It is a Maven-based project.

Resources Committed (developers committed to working)

* Allan Clarke, @allanclarke, aclarke@luminanetworks.com, Lumina Networks
* Balaji Varadaraju, bvaradar@luminanetworks.com, Lumina Networks
* Tejas Nevrekar, tnevrekar@luminanetworks.com, Lumina Networks

Initial Committers

This is the upstreaming of an existing proprietary code base.

* Allan Clarke, @allanclarke, aclarke@luminanetworks.com, Lumina Networks
* Balaji Varadaraju, bvaradar@luminanetworks.com, Lumina Networks
* Tejas Nevrekar, tnevrekar@luminanetworks.com, Lumina Networks

Vendor Neutral

The following has/will be done to the code base prior to upstreaming:
* Java packages have had the vendor removed
* All copyrights have been updated to EPL 1.0
* All other references to the vendor have been removed from the tutorial, documentation, and code
* Maven dependencies have been updated to use
* Modularity conventions will be folded in over the initial weeks of the project's lifetime as needed

Meets Board Policy (including IPR)

TBD